

## Recursive Algorithms

You know that a function can call another function; that is, a statement in the body of a function  $F$  contains a call of another function  $G$ . What would happen if a statement in  $F$  contained a call of  $F$ ?

This situation—a function or procedure containing a call on itself—is not only permitted but in fact is very interesting and useful. It is, in fact, a mathematical concept called *recursion*, and a subprogram that contains a call to itself is called a recursive subprogram.

In many instances the use of recursion enables us to specify a natural, simple solution to a problem that would otherwise be difficult to solve. For this reason, recursion is an important and powerful tool in problem solving and programming.

# The Nature of Recursion

Recursive solutions have the following characteristics:

- One or more simple cases of the problem (called *stopping cases*) have a simple, nonrecursive solution.
- For the other cases, there is a process (using recursion) for substituting one or more reduced cases of the problem that are closer to a stopping case.
- Eventually the problem can be reduced to stopping cases only, all of which are relatively easy to solve.

The recursive algorithms that we write will generally consist of an if statement with the form shown below:

```
if (the stopping case is reached)
    Solve it
else
    Reduce the problem using recursion
```

# Recursion Examples

- Multiplication

```
public static int Multiply (int M, int N)
{
    // Performs multiplication recursively using the + operator
    // Pre : M and N are defined and N > 0
    // Post: returns M * N

    int Result;

    if (N == 1)
    {
        Result = M;           // stopping case
    }
    else
    {
        Result = M + Multiply(M, N-1); // recursion
    }

    return Result;
}
```

```

//-----
// TestMultiply.java
// Illustrates recursive Multiply algorithm
// Author: M.B. Feldman, The George Washington University
// Last Modified: March 2003
//-----
import cs1.Keyboard;
public class TestMultiply
{
    public static int Multiply (int M, int N)
    {
        // Performs multiplication recursively using the + operator
        // Pre : M and N are defined and N > 0
        // Post: returns M * N

        int Result;

        if (N == 1)
        {
            Result = M;           // stopping case
        }
        else
        {
            Result = M + Multiply(M, N-1); // recursion
        }

        return Result;
    }
}

```

```
public static void main (String[] args)
{
    for (int Num1 = 1; Num1 <= 5; Num1++)
    {
        for (int Num2 = 1; Num2 <= 5; Num2++)
        {
            System.out.print(Multiply(Num1, Num2) + "\t");
        }
        System.out.println();
    }
}
```

- Factorial

```
public static int Factorial (int N)
{
    // Computes the factorial of N (N!) recursively
    // Pre : N is defined
    // Post: returns N!

    if (N == 1)
    {
        return 1;                // stopping case
    }
    else
    {
        return N * Factorial(N-1); // recursion
    }
}
```

```

//-----
// TestFactorial.java
// Illustrates recursive Factorial algorithm
// Author: M.B. Feldman, The George Washington University
// Last Modified: March 2003
//-----
import cs1.Keyboard;
public class TestFactorial
{
    public static int Factorial (int N)
    {
        // Computes the factorial of N (N!) recursively
        // Pre : N is defined
        // Post: returns N!

        if (N == 1)
        {
            return 1;                // stopping case
        }
        else
        {
            return N * Factorial(N-1); // recursion
        }
    }
}

```

```
public static void main (String[] args)
{
    System.out.println("  N\tN!");
    System.out.println();

    for (int Num = 1; Num <= 20; Num++)
    {
        System.out.println(Num + "\t" + Factorial(Num));
    }
}
}
```

- Fibonacci Numbers

A certain man put a pair of rabbits in a place surrounded on all sides by a wall. How many pairs of rabbits can be produced from that pair in a year if it is supposed that every month each pair begets a new pair which from the second month on becomes productive?

```
public static int Fibonacci (int N)
{
    // Returns the Nth Fibonacci number, computed recursively
    // Pre : N is defined and N >= 0
    // Post: returns N!

    if ((N == 1) || (N == 2))
    {
        return 1;
    }
    else
    {
        return Fibonacci(N-2) + Fibonacci(N-1);
    }
}
```

- Greatest Common Divisor

The greatest common divisor (GCD) of two positive integers is the largest integer that divides them both. The ancient mathematician Euclid (Alexandria, Egypt, 325-265 BC) discovered this algorithm for finding the GCD:

```
public static int GCD (int M, int N)
{
    int Result;

    if ((N <= M) && (M % N == 0))
    {
        Result = N;                // stopping case
    }
    else if (M < N)
    {
        Result = GCD(N, M);       // recursion
    }
    else
    {
        Result = GCD(N, M % N);   // recursion
    }
    return Result;
}
```

- String Reverse

```
//-----  
// Finds reverse of a string, recursively  
//-----  
public static String reverse (String S)  
{  
    System.out.println("Now examining " + S);  
    if (S.length() <= 1)  
    {  
        return S;                // stopping case  
    }  
    else  
    {  
        return reverse(S.substring(1)) + S.charAt(0);  
        // recursion  
    }  
}
```

```

//*****
//  TestPalindrome.java    M. Feldman, derived from Lewis/Loftus
//
//  Demonstrates the use of recursive string reverse method
//*****

import cs1.Keyboard;

public class TestPalindrome
{
    //-----
    //  Finds reverse of a string, recursively
    //-----
    public static String reverse (String S)
    {
        System.out.println("Now examining " + S);
        if (S.length() <= 1)
        {
            return S;                // stopping case
        }
        else
        {
            return reverse(S.substring(1)) + S.charAt(0);
                                   // recursion
        }
    }
}

```

```

//-----
// Tests strings to see if they are palindromes.
//-----
public static void main (String[] args)
{
    String str, rev, another = "y";

    while (another.equalsIgnoreCase("y")) // allows y or Y
    {
        System.out.print ("Enter a potential palindrome: ");
        str = Keyboard.readString();

        rev = reverse(str);

        System.out.println("Its reverse is " + rev);

        if (rev.equals(str))
        {
            System.out.println ("That string IS a palindrome.");
        }
        else
        {
            System.out.println
                ("That string IS NOT a palindrome.");
        }
    }
}

```

```
System.out.println();  
System.out.print ("Test another palindrome (y/n)? ");  
another = Keyboard.readString();
```

```
}
```

```
}
```

```
}
```

## • Binary Search

```
// Main, Fig. 11.2, p. 554
public static int
  search (int[] a, int first, int size, int target)
{
  int middle;

  if (size <= 0)
    return -1;
  else
  {
    middle = first + size/2;
    if (target == a[middle])
      return middle;
    else if (target < a[middle])
      // the target is less than a[middle],
      // so search before the middle
      return search(a, first, size/2, target);
    else
      // the target must be greater than a[middle],
      // so search after the middle
      return search(a, middle+1, (size-1)/2, target);
  }
}
```